Attorney Docket No.: 06502.0376

Express Mail Label No.: EL535776915US

UNITED STATES PATENT APPLICATION

OF

GUY L. STEELE, JR.

FOR

FLOATING POINT SYSTEM WITH IMPROVED SUPPORT OF INTERVAL ARITHMETIC

INCORPORATION BY REFERENCE

[001] Related U. S. Patent Application Serial No. not yet assigned, filed on even date herewith in the name of Guy L. Steele, Jr. and entitled "Floating Point System That Represents Status Flag Information Within A Floating Point Operand," assigned to the assignee of the present application, is hereby incorporated by reference.

[002] Related U. S. Patent Application Serial No. not yet assigned, filed on even date herewith in the name of Guy L. Steele, Jr. and entitled "Floating Point Adder In Which Floating Point Status Information is Encoded in Floating Point Representations," assigned to the assignee of the present application, is also hereby incorporated by reference.

[003] Related U. S. Patent Application Serial No. not yet assigned, filed on even date herewith in the name of Guy L. Steele, Jr. and entitled "Floating Point Multiplier In Which Floating Point Status Information is Encoded in Floating Point Representations," assigned to the assignee of the present application, is also hereby incorporated by reference.

[004] Related U. S. Patent Application Serial No. not yet assigned, filed on even date herewith in the name of Guy L. Steele, Jr. and entitled "Floating Point Divider In Which Floating Point Status Information is Encoded in Floating Point Representations," assigned to the assignee of the present application, is also hereby incorporated by reference.

FIELD OF THE INVENTION

[005] The invention relates generally to systems and methods for performing floating point operations, and more particularly to systems, methods, and instructions for performing arithmetic operations on floating point operands while enhancing support for interval arithmetic.

BACKGROUND OF THE INVENTION

Floating Point Arithmetic

[006] Digital electronic devices, such as digital computers, calculators, and other devices, perform arithmetic calculations on values in integer, or "fixed point," format, in fractional, or "floating point" format, or both. IEEE Standard 754, (hereinafter "IEEE Std. 754" or "the Standard") published in 1985 by the Institute of Electrical and Electronic Engineers, and adopted by the American National Standards Institute (ANSI), defines several standard formats for expressing values in floating point format, and a number of aspects regarding the behavior of computations in connection therewith. In accordance with IEEE Std. 754, a representation in floating point format comprises a plurality of binary digits, or "bits," having the structure

$$se_{msb}\ldots e_{lsb}\, f_{msb}\ldots f_{lsb}$$

where bit "s" is a sign bit indicating whether the entire value is positive or negative, bits "$e_{msb}\ldots e_{lsb}$" comprise an exponent field representing the exponent "e" in unsigned binary biased format, and bits "$f_{msb}\ldots f_{lsb}$" comprise a fraction field that represents the fractional portion "f" in unsigned binary format ("msb" represents "most significant bit" and "lsb" represents "least significant bit"). The Standard defines two general formats, namely, a "single" format which comprises thirty-two bits, and a "double" format which comprises sixty-four bits. In the single format, there is one sign bit "s," eight bits "$e_7\ldots e_0$" comprising the exponent field and twenty-three bits "$f_{22}\ldots f_0$" comprising the fraction field. In the double format, there is one sign bit "s," eleven bits "$e_{10}\ldots e_0$" comprising the exponent field and fifty-two bits "$f_{51}\ldots f_0$" comprising the fraction field.

[007] As indicated above, the exponent field of the floating point representation "$e_{msb}\ldots e_{lsb}$" represents the exponent "E" in biased format. The biased format provides a

mechanism by which the sign of the exponent is implicitly indicated. In particular, the bits "$e_{msb}...e_{lsb}$" represent a binary encoded value "e" such that "e=E+bias." This allows the exponent E to extend from -126 to +127, in the eight-bit "single" format, and from -1022 to +1023 in the eleven-bit "double" format, and provides for relatively easy manipulation of the exponents in multiplication and division operations, in which the exponents are added and subtracted, respectively.

[008] IEEE Std. 754 provides for several different formats with both the single and double formats, which are generally based on the bit patterns of the bits $e_{msb}...e_{lsb}$ comprising the exponent field and the bits $f_{msb}...f_{lsb}$ comprising the fraction field. As shown in prior art FIG. 1, if a number is represented such that all of the bits $e_{msb}...e_{lsb}$ of the exponent field are binary ones (that is, if the bits represent a binary-encoded value of "255" in the single format or "2047" in the double format) and all of the bits $f_{msb}...f_{lsb}$ of the fraction field are binary zeros, then the value of the number is positive infinity 110 or negative infinity 120, depending on the value of the sign bit "s." In particular, the value "v" is $v = (-1)^s\infty$ where "∞" represents the value "infinity." On the other hand, if the bit pattern is formatted such that all of the bits $e_{msb}...e_{lsb}$ of the exponent field are binary ones and the bits $f_{msb}...f_{lsb}$ of the fraction field are not all zeros, then the value that is represented is deemed "not a number," abbreviated in the Standard by "NaN" 130.

[009] If a number has an exponent field in which the bits $e_{msb}...e_{lsb}$ are neither all binary ones nor all binary zeros (that is, if the bits represent a binary-encoded value between 1 and 254 in the single format or between 1 and 2046 in the double format), the number is said to be in a "normalized" format 160. For a number in the normalized format, the value represented by the number is $v = (-1)^s 2^{e-bias}(1.|f_{msb}...f_{lsb})$, where "|" represents a

concatenation operation. Effectively, in the normalized format, there is an implicit most significant digit having the value "one," so that the twenty-three digits in the fraction field of the single format, or the fifty-two digits in the fraction field of the double format, will effectively represent a value having twenty-four digits or fifty-three digits of precision, respectively, where the value is less than two, but not less than one.

[010] On the other hand, if a number has an exponent field in which the bits $e_{msb}...e_{lsb}$ are all binary zeros, representing the binary-encoded value of "zero," and a fraction field in which the bits $f_{msb}...f_{lsb}$ are not all zero, the number is said to be in a "denormalized" format 170. For a number in the denormalized format, the value represented by the number is $v = (-1)^s 2^{e-bias+1}(0.|f_{msb}...f_{lsb})$. It will be appreciated that the range of values of numbers that can be expressed in the denormalized format is disjoint from the range of values of numbers that can be expressed in the normalized format, for both the single and double formats. Finally, if a number has an exponent field in which the bits $e_{msb}...e_{lsb}$ are all binary zeros, representing the binary-encoded value of "zero," and a fraction field in which the bits $f_{msb}...f_{lsb}$ are all zero, the number has the value "zero." It will be appreciated that the value "zero" may be positive zero 140 or negative zero 150, depending on the value of the sign bit.

[011] Generally, floating point units to perform computations whose results conform to IEEE Std. 754 are designed to generate a result in response to a floating point instruction in three steps:

[012] (a) First, an approximation calculation step in which an approximation to the absolutely accurate mathematical result (assuming that the input operands represent the specific mathematical values as described by IEEE Std. 754) is calculated that is sufficiently precise as to allow this accurate mathematical result to be summarized by a sign bit, an

exponent (typically represented using more bits than are used for an exponent in the standard floating-point format), and some number "N" of bits of the presumed result fraction, plus a guard bit and a sticky bit. The value of the exponent will be such that the value of the fraction generated in step (a) consists of a 1 before the binary point and a fraction after the binary point. The bits are calculated so as to obtain the same result as the following conceptual procedure (which is impossible under some circumstances to carry out in practice): calculate the mathematical result to an infinite number of bits of precision in binary scientific notation, and in such a way that there is no bit position in the significand such that all bits of lesser significance are 1-bits (this restriction avoids the ambiguity between, for example, 1.100000... and 1.011111... as representations of the value "one-and-one-half"); then let the N most significant bits of the infinite significand be used as the intermediate result significand, let the next bit of the infinite significand be the guard bit, and let the sticky bit be 0 if and only if ALL remaining bits of the infinite significand are 0-bits (in other words, the sticky bit is the logical OR of all remaining bits of the infinite fraction after the guard bit).

[013] (b) Second, a rounding step, in which the guard bit, the sticky bit, perhaps the sign bit, and perhaps some of the bits of the presumed significand generated in step (a) are used to decide whether to alter the result of step (a). For the rounding modes defined by IEEE Std. 754, this is a decision as to whether to increase the magnitude of the number represented by the presumed exponent and fraction generated in step (a). Increasing the magnitude of the number is done by adding 1 to the significand in its least significant bit position, as if the significand were a binary integer. It will be appreciated that, if the significand is all 1-bits, then magnitude of the number is "increased" by changing it to a

high-order 1-bit followed by all 0-bits and adding 1 to the exponent. It will be further appreciated that,

[014] (i) if the result is a positive number, and

[015] (a) if the decision is made to increase, effectively the decision has been made to increase the value of the result, thereby rounding the result up, towards positive infinity, but

[016] (b) if the decision is made not to increase, effectively the decision has been made to decrease the value of the result, thereby rounding the result down, towards negative infinity; and

[017] (ii) if the result is a negative number, and

[018] (a) if the decision is made to increase, effectively the decision has been made to decrease the value of the result, thereby rounding the result down, towards negative infinity, but

[019] (b) if the decision is made not to increase, effectively the decision has been made to increase the value of the result, thereby rounding the result up, towards positive infinity.

[020] (c) Finally, a packaging step, in which the result is packaged into a standard floating-point format. This may involve substituting a special representation, such as the representation format defined for infinity or NaN if an exceptional situation (such as overflow, underflow, or an invalid operation) was detected. Alternatively, this may involve removing the leading 1-bit (if any) of the fraction, because such leading 1-bits are implicit in the standard format. As another alternative, this may involve shifting the fraction in order to construct a denormalized number. As a specific example, we assume that this is the step that

forces the result to be a NaN if any input operand is a NaN. In this step, the decision is also made as to whether the result should be an infinity. It will be appreciated that, if the result is to be a NaN or infinity, any result from step (b) will be discarded and instead the appropriate representation in the appropriate format will be provided as the result.

[021] In addition in the packaging step, floating-point status information is generated, which is stored in a floating point status register. The floating point status information generated for a particular floating point operation includes indications, for example, as to whether:

[022] (i) a particular operand is invalid for the operation to be performed ("invalid operation");

[023] (ii) if the operation to be performed is division, the divisor is zero ("division-by-zero");

[024] (iii) an overflow occurred during the operation ("overflow");

[025] (iv) an underflow occurred during the operation ("underflow"); and

[026] (v) the rounded result of the operation is not exact ("inexact").

[027] These conditions are typically represented by flags that are stored in the floating point status register, separate from the floating point operand. The floating point status information can be used to dynamically control the operations performed in response to certain instructions, such as conditional branch, conditional move, and conditional trap instructions that may be in the instruction stream subsequent to the floating point instruction. Also, the floating point status information may enable processing of a trap sequence, which will interrupt the normal flow of program execution. In addition, the floating point status information may be used to affect certain ones of the functional unit control signals that

control the rounding mode. IEEE Std. 754 also provides for accumulating floating point status information from, for example, results generated for a series or plurality of floating point operations.

[028] IEEE Std. 754 has brought relative harmony and stability to floating-point computation and architectural design of floating-point units. Moreover, its design was based on some important principles and rests on sensible mathematical semantics that ease the job of programmers and numerical analysts. It also provides some support for the implementation of interval arithmetic, which may prove to be preferable to simple scalar arithmetic for many tasks. Nevertheless, IEEE Std. 754 has some serious drawbacks, including:

[029] (i) Modes, which include the rounding mode and may also include a traps enabled/disabled mode, flags representing the floating point status information that is stored in the floating point status register, and traps that are required to implement IEEE Std. 754, all introduce implicit serialization between floating-point instructions, and between floating point instructions and the instructions that read and write the flags and modes. Implicit serialization occurs when programmers and designers try to avoid the problems caused if every floating point instructions uses, and can change, the same floating point status register. This can create problems if, for example, two instructions are executing in parallel in a microprocessor architectures featuring several CPUs running at once and both instructions cause an update of the floating point status register. In such a case, the contents of the status register would likely be incorrect with respect to at least one of the instructions, because the other parallel instruction will have written over the original contents. Similar problems can occur in scalar processor architectures, in which several instructions are issued and processed

at once. To solve this problem, programmers and designers serialize floating point instructions that can affect the floating point status register, making sure they execute in a serial fashion, one instruction completing before another begins. Rounding modes can introduce implicit serialization because they are typically indicated as global state, although in some microprocessor architectures, the rounding mode is encoded as part of the instruction operation code, which will alleviate this problem to that extent. This implicit serialization makes the Standard difficult to implement coherently in today's superscalar and parallel microprocessor architectures without loss of performance.

[030] (ii) The implicit side effects of a procedure that can change the flags or modes can make it very difficult for compilers to perform optimizations on floating-point code; to be safe, compilers for most languages must assume that every procedure call is an optimization barrier.

[031] (iii) Global flags, such as those that signal certain modes, make it more difficult to do instruction scheduling where the best performance is provided by interleaving instructions of unrelated computations. Instructions from regions of code governed by different flag settings or different flag detection requirements cannot easily be interleaved when they must share a single set of global flag bits in a global floating point status register.

[032] (iv) Traps have been difficult to integrate efficiently into architectures and programming language designs for fine-grained control of algorithmic behavior.

[033] U.S. Patent Application Serial No. __not yet assigned__, filed on even date herewith in the name of Guy L. Steele Jr. and entitled "Floating Point Unit In Which Floating Point Status Information Is Encoded In Floating Point Representations," describes a floating point unit in which floating point status information is encoded in the representations of the

results generated thereby. By encoding the floating point status information relating to a floating point operation in the result that is generated for the operation, the implicit serialization required by maintaining the floating point status information separate and apart therefrom can be obviated. The floating point unit includes a plurality of functional units, including an adder unit, a multiplier unit, a divider unit, a square root unit, a maximum/minimum unit, a comparator unit and a tester unit, all of which operate under control of functional unit control signals provided by a control unit. It may also include features consistent with the principles of the present invention to provide better support for interval arithmetic.

Interval Arithmetic

[034] Interval arithmetic can be used when processing floating point operands to compute an interval result. In general, an interval is the set of all real numbers between and including the lower and upper bound of the interval. Interval arithmetic is used to evaluate arithmetic expressions over sets of numbers contained in intervals. An interval arithmetic result is a new interval that contains the set of all possible resulting values.

[035] In interval arithmetic computations, each computed value is represented as a pair of numbers [a, b] (where a <= b) that indicates a lower bound and an upper bound of an interval on the real number line. One can regard an interval [a, b] as a set:

$$\{ p \mid a <= p <= b \}$$

[036] It is also convenient to use "-Infinity" and "+Infinity" as bounds. For example, [3, +Infinity] represents the set of all numbers not less than 3.

[037] Those skilled in the art will recognize that the most general definition of a binary operation **f** on two intervals **x** and **y** in terms of an underlying binary operation (also called **f**) on real numbers is:

$$[\ \text{glb } S, \text{lub } S\ ] \text{ where } S = \{\ f(p,q) \mid p \text{ in } x \text{ and } q \text{ in } y\ \}$$

[038] That is, one considers all possible pairs of real arguments where each argument ranges over an input interval and computes **f** on all these possible pairs; the result is a set **S** of real results. The set **S** may fail to be contiguous, so to produce an interval result it is necessary to "fill it in." The greatest lower bound (glb) of **S** and the least upper bound (lub) of **S** may be used as endpoints of the result interval.

[039] For particular "well-behaved" binary operations, it is not necessary to consider an infinite set of computations to compute the endpoints of the result interval. In particular, intervals can be added, subtracted, multiplied, and divided using the formulae:

$$[a,b]+[c,d]=[a+c,b+d]$$
$$[a,b]-[c,d]=[a-d,b-c]$$
$$[a,b]*[c,d]=[\min(a*c,\ a*d,\ b*c,\ b*d),\ \max(a*c,\ a*d,\ b*c,\ b*d)]$$
$$[a,b]\ /\ [c,d]=[\min(a/c,\ a/d,\ b/c,\ b/d),\ \max(a/c,\ a/d,\ b/c,\ b/d)]\ \text{provided } c>0 \text{ or } d<0$$

[040] If the numbers are represented in a floating-point representation with finite precision, then addition and multiplication of such numbers produce only approximate results. However, by controlling the rounding of such results, floating-point arithmetic can be correctly used for interval computations. The typical consequence of the floating-point approximations is only that result intervals may be slightly larger than mathematically necessary.

[041] Standard IEEE 754 supports an implementation of interval arithmetic by providing rounding modes such as "round toward plus infinity" and "round toward minus infinity."

[042] For purposes of the following discussion, let "+UP" denote floating-point addition using rounding mode "round toward plus infinity." Let "-UP" denote floating-point subtraction using rounding mode "round toward plus infinity." Let "*UP" denote floating-point multiplication using rounding mode "round toward plus infinity." Let "/UP" denote floating-point division using rounding mode "round toward plus infinity." Let "+DOWN" denote floating-point addition using rounding mode "round toward minus infinity." Let "- DOWN" denote floating-point subtraction using rounding mode "round toward minus infinity." Let "*DOWN" denote floating-point multiplication using rounding mode "round toward minus infinity." Finally, let "/DOWN" denote floating-point division using rounding mode "round toward minus infinity."

[043] Under these naming conventions, the interval computation rules using floating-point arithmetic may be written as:

$$[a, b] + [c, d] = [a +_{DOWN} c,\ b +_{UP} d]$$

$$[a, b] - [c, d] = [a -_{DOWN} d,\ b -_{UP} c]$$

$$[a, b] * [c, d] = [\min(a *_{DOWN} c,\ a *_{DOWN} d,\ b *_{DOWN} c,\ b *_{DOWN} d),\ \max(a *_{UP} c,\ a *_{UP} d,\ b *_{UP} c,\ b *_{UP} d)]$$

$$[a, b] / [c, d] = \text{if } (c > 0 \text{ or } d < 0) \text{ then } [\min(a /_{DOWN} c,\ a /_{DOWN} d,\ b /_{DOWN} c,\ b /_{DOWN} d),\ \max(a /_{UP} c,\ a /_{UP} d,\ b /_{UP} c,\ b /_{UP} d)] \text{ else } [-\text{Infinity}, +\text{Infinity}].$$

[044] Despite this interval arithmetic support, IEEE 754-1985 has some drawbacks for performing interval arithmetic operations. There are certain situations in which IEEE 754 specifies that a floating-point operation must deliver a result in the NaN format, even though

the rounding mode is "round toward plus infinity" or "round toward minus infinity" and even though a reasonable result could be returned that would be useful as an interval bound. For example, consider that two intervals:

[-Inf, 6] and [+Inf, +Inf].

The sum of these two intervals may be computed as:

[-Inf +DOWN +Inf, 6 +UP +Inf].

where "Inf" means "infinity" as specified by IEEE 754. It is clear that a reasonable result would be:

[-Inf, +Inf].

[045] However, IEEE 754 specifies that, under any rounding mode, the result of (-Inf) + (+Inf) is a NaN result. Therefore, using IEEE 754 arithmetic to compute this interval sum produces:

[NaN, +Inf],

which is not a properly formed interval. This may undesirably lead to confusing or invalid results for interval arithmetic operations and, potentially, to erroneous calculations.

[046] To prevent malformed intervals such as in the previous example, programmers of IEEE 754 machines typically design special software routines to recognize inputs that create such malformed intervals and use extra code routines to handle them before doing the interval arithmetic operation. Disadvantageously, such special routines waste time and space in software programs.

Thus, there is a need for systems and methods that efficiently support addition, subtraction, multiplication, and division of intervals using floating-point arithmetic similar to

that of IEEE 754, but such that the result of every interval operation is a well-formed interval.

SUMMARY OF THE INVENTION

[047] Systems and methods according to the principles of the present invention apply to IEEE 754 format machines and systems, as well as machines and systems that have floating point status information encoded in their operands and results, as described in the related applications.

[048] Embodiments consistent with the principles of the present invention provide improved results, compared to IEEE Std. 754, for floating point operations used in interval arithmetic calculations. One embodiment consistent with the principles of the present invention provides a method of enhancing support of an interval computation when performing a floating point arithmetic operation, comprising the steps, performed by a processor, of receiving a first floating point operand, receiving a second floating point operand, executing the floating point arithmetic operation on the first floating point operand and the second floating point operand, determining whether a NaN substitution is necessary, producing a floating point result if the NaN substitution is determined to be unnecessary, and substituting an alternative value as the floating point result if the NaN substitution is determined to be necessary.

BRIEF DESCRIPTION OF THE DRAWINGS

[049] This invention is pointed out with particularity in the appended claims. The above and further advantages of this invention may be better understood by referring to the following description taken in conjunction with the accompanying drawings, in which:

[050] FIG. 1 depicts prior art formats for representation of floating point values;

[051] FIG. 2 is a table showing the IEEE Standard 754 results compared to exemplary improved interval arithmetic results consistent with the principles of the present invention, for various floating point arithmetic operations;

[052] FIG. 3 is a functional block diagram of an exemplary adder/subtracter unit according to one embodiment of the invention;

[053] FIG. 4 is a functional block diagram of an exemplary multiplier unit according to one embodiment of the invention; and

[054] FIG. 5 is a functional block diagram of an exemplary divider unit according to one embodiment of the invention.

DETAILED DESCRIPTION OF AN ILLUSTRATIVE EMBODIMENT

[055] The following description of embodiments consistent with the principles of the present invention assumes 32-bit floating-point numbers. However, adapting it to 64 bits or to other sizes is straightforward and obvious to one who is of ordinary skill in the art. The embodiments described contemplate using general purpose computer instructions to implement the invention, but one of ordinary skill will recognize that specialized interval arithmetic computer instructions, or other designs, could also easily be used to implement the invention.

[056] Systems and methods consistent with the principles of the present invention provide support for floating-point arithmetic in the manner of IEEE 754 and in the manner described in related, incorporated-by-reference U. S. Patent Application Serial No. not yet assigned, filed on even date herewith in the name of Guy L. Steele, Jr. entitled "Floating Point System That Represents Status Flag Information Within A Floating Point Operand," assigned to the assignee of the present application. Embodiments consistent with the

principles of the present invention define results for arithmetic operations with the rounding modes "round toward plus infinity" and "round toward minus infinity" such that a "NaN" is not produced as a result if none of the inputs is a NaN. Eliminating NaN as a result improves interval arithmetic calculations because valid intervals are produced instead of malformed intervals with NaN as an endpoint. Further, substituting a non-NaN result for the conventional NaN result allows for efficient processing because execution time and code space are not wasted on special routines otherwise used to handle a conventional NaN result. Systems and methods consistent with the principles of the present invention may comprise software to be used for performing addition, subtraction, multiplication, and division of intervals on a computer system. Software that supports the improved definition of floating-point addition, subtraction, multiplication, and division, performs interval computations that produce more precise and reliable results than computations performed by existing systems.

[057] One embodiment consistent with the principles of the invention supports floating-point arithmetic as defined by related U. S. Patent Application Serial No. not yet assigned, filed on even date herewith in the name of Guy L. Steele, Jr. entitled "Floating Point System That Represents Status Flag Information Within A Floating Point Operand," assigned to the assignee of the present application. In this embodiment, the floating point operands have status information encoded within the operand itself. Additionally, the floating point arithmetic includes computations involving positive and negative overflow and underflow values (+OV, -OV, +UN, and -UN). Another embodiment consistent with the principles of the invention supports IEEE 754-compliant arithmetic, except for the changes in the handling of zero and infinity operands as described herein.

[058] Related U. S. Patent Application Serial No. not yet assigned, filed on even date herewith in the name of Guy L. Steele, Jr. and entitled "Floating Point Adder In Which Floating Point Status Information is Encoded in Floating Point Representations," assigned to the assignee of the present application, describes improved add and subtract instructions. In brief summary, subtraction is defined to behave as (a - b) = (a + (-b)), where unary "-" simply flips the sign bit of its operand. As shown in FIG. 2, an embodiment consistent with the principles of the present invention improves addition operations 205 and subtraction operations 207 by providing that for "round toward plus infinity" the result of adding infinities of opposite signs is +Infinity, substituted for NaN as specified by IEEE Std. 754. Similarly, for "round toward minus infinity" addition 206 and subtraction 208 operations, the result of adding infinities of opposite signs is -Infinity, instead of NaN. As a result, the standard formulae for interval addition and subtraction:

$$[a,\ b] + [c, d] = [a\ +\text{DOWN}\ c,\ b\ +\text{UP}\ d]$$
$$[a,\ b] - [c, d] = [a\ -\text{DOWN}\ d,\ b\ -\text{UP}\ c]$$

produce improved results in certain situations where IEEE 754 would produce a malformed interval (e.g., an interval with a NaN) or would necessitate the use of a more complicated formula to forestall the production of a NaN result.

[059] Related U. S. Patent Application Serial No. not yet assigned, filed on even date herewith in the name of Guy L. Steele, Jr. and entitled "Floating Point Multiplier In Which Floating Point Status Information is Encoded in Floating Point Representations," assigned to the assignee of the present application, describes an improved multiplication instruction. An embodiment consistent with the principles of the present invention improves multiplication operations by providing that a multiplication with "round toward plus infinity" 210 produces -0 as the result of multiplying an infinity and a zero of opposite signs,

substituting for NaN as specified by IEEE 754. Similarly, the result of multiplying an infinity and a zero of like signs is forced to +Infinity, instead of resulting in a NaN. Also, for "round toward minus infinity" multiplication 215, the result of multiplying an infinity and a zero of opposite signs is -Infinity, not NaN, and the result of multiplying an Infinity and a zero of like signs is +0, not NaN. Using such an implementation, the standard formula for interval multiplication:

$$[a, b] * [c, d] = [\min(a *_{DOWN} c, a *_{DOWN} d, b *_{DOWN} c, b *_{DOWN} d), \max(a *_{UP} c, a *_{UP} d, b *_{UP} c, b *_{UP} d)]$$

produces improved results in certain situations where IEEE 754 would otherwise produce a malformed interval (e.g., an interval containing a NaN) or would necessitate the use of a more complicated formula to forestall the production of a NaN result.

[060] Related U. S. Patent Application Serial No. __not yet assigned__, filed on even date herewith in the name of Guy L. Steele, Jr. and entitled "Floating Point Divider In Which Floating Point Status Information is Encoded in Floating Point Representations," assigned to the assignee of the present application, describes an improved division instruction. An embodiment consistent with the principles of the present invention improves divide operations by providing that for "round toward plus infinity" division 220, the result of dividing two infinities or two zeros of opposite signs is -0, substituted for NaN as specified by IEEE 754. Additionally, the result of dividing two infinities or two zeros of like signs is +Infinity, instead of NaN. Similarly, for "round toward minus infinity" division 225, the result of dividing two infinities or two zeros of opposite signs is -Infinity, not NaN, and the result of dividing two infinities or two zeros of like sign is +0, substituted for NaN. Using such an improvement, the standard formula for interval division:

$$[a, b] / [c, d] = \text{if } (c > 0 \text{ or } d < 0) \text{ then}$$
$$[\min(a /\text{DOWN } c, a /\text{DOWN } d, b *\text{DOWN } c, b *\text{DOWN } d),$$
$$\max(a /\text{UP } c, a /\text{UP } d, b /\text{UP } c, b /\text{UP } d)]$$
$$\text{else } [-\text{Infinity}, +\text{Infinity}]$$

produces improved results in certain situations where IEEE 754 would produce a malformed interval (e.g., an interval containing a NaN) or would necessitate the use of a more complicated formula to forestall the production of a NaN result.

[061] FIG. 3 is a functional block diagram of an exemplary adder/subtracter unit according to one embodiment of the invention. In summary, the exemplary adder/subtracter unit 310 receives two floating point operands and generates a result, and in some cases floating point status information, based on the operands. In the embodiment shown, the floating point status information is encoded within the floating point representation of the result. Having the floating point status information comprise part of the result, instead of being separate and apart from the result, (for example in a separate floating point status register), obviates the implicit serialization of floating point instructions that would be otherwise required.

[062] The exemplary adder/subtracter unit 310 includes two operand buffers 311A and 311B, respective operand analysis circuits 312A and 312B, an adder core 313, a result assembler 314, and an adder decision table logic circuit 315. The operand buffers 311A and 311B receive and store respective operands that may be received from, for example, a set of registers (not shown) in a conventional manner.

[063] Generally, the adder core 313 receives the operands from the operand buffers 311A and 311B and receives rounding mode information from, for example, a rounding mode store 316. Rounding mode store 316 supplies signals indicating, for example, whether

the rounding mode for the current operation is "round toward plus infinity" or "round toward minus infinity." In accordance with the principles of the present invention, one embodiment of the adder unit 310 generates results according to the table in FIG. 2, as discussed above. In one embodiment, adder core 313 produces results conforming to IEEE Standard 754; certain addition and subtraction operations always produce a NaN result regardless of the rounding mode, thus leading to production of malformed intervals if left unmodified. In the embodiment shown, however, adder decision table logic 315, which controls the result assembler 314, replaces a NaN result with a non-NaN result. In one embodiment, adder decision table logic 315 determines whether to substitute a non-NaN result for a NaN result by examining the first operand 311A and the second operand 311B before or during the operation of adder core 313, knowing that certain addition operands produce a NaN under IEEE 754. In such an embodiment, adder core 313 could be simplified to not handle cases where a NaN result would be replaced by adder decision table logic 315 (making the core IEEE 754 noncompliant), or adder core 313 could be bypassed so as not to operate at all in cases where a NaN result would be replaced. In another embodiment that is probably slower because the replacement determination is not made in parallel with the adder core's operation, adder decision table logic 315 determines whether to substitute a non-NaN result for a NaN result by examining the result produced by adder core 313. In yet another embodiment easily within the knowledge of one of ordinary skill in the art, adder core 313 produces a non-NaN result for cases where IEEE 754 compliance would otherwise call for a NaN result. In this embodiment, the core in effect self-determines and replaces an IEEE 754 NaN result with a non-NaN result, obviating the need for adder decision table logic 315 to

make the determination and replacement. In such an embodiment, adder core 313 would obviously no longer be IEEE 754 compliant.

[064] Each operand analysis circuit 312A, 312B analyzes the operand in the respective operand buffer 311A, 311B and generates signals providing information relating to the respective operands. These signals are provided to the adder decision table logic 315.

[065] The result assembler 314 receives information from a number of sources, including the operand buffers 311A and 311B, adder core 313, and several predetermined value stores. Under control of signals from the adder decision table logic 315, result assembler 314 generates the result and provides it to a result bus 317. The result bus 317 may provide the result signals to any convenient destination, such as a register or memory unit, for storage or other use.

[066] One of ordinary skill in the art will recognize that an improved adder core designed to produce results according to the table of FIG. 2 could be implemented in a floating point adder/subtracter unit that does not use floating point status information encoded within the floating point operands, without departing from the principles of the present invention. One of ordinary skill will also realize that the principles of the present invention are also applicable to conventional floating point adder/subtracter architectures that, for example, maintain floating point status information in a global floating point status register, separate from the operands and result. Moreover, it would also be apparent to one of ordinary skill in the art that the NaN result substitutions according to the table in FIG. 2 are not the only possible logical substitutions. Other non-NaN results could be used within the scope of the present invention.

[067] FIG. 4 is a functional block diagram of an exemplary multiplier unit according to one embodiment of the invention. Generally, the exemplary multiplier unit 410 receives two floating point operands and generates therefrom a result. In some cases, multiplier unit 410 also generates floating point status information, which is encoded in and comprises part of the floating point representation of the result. Since the floating point status information comprises part of the floating point representation of the result, instead of being separate and apart from the result as in conventional multiplier units, the implicit serialization that is required by maintaining the floating point status information separate and apart from the result can be obviated.

[068] As shown in FIG. 4, the exemplary multiplier unit 410 includes two operand buffers 411A and 411B, respective operand analysis circuits 412A and 412B, a multiplier core 413, a result assembler 414 and an multiplier decision table logic circuit 415. The operand buffers 411A and 411B receive and store respective operands from, for example, a set of registers (not shown) or memory (not shown) in a conventional manner.

[069] Generally, the multiplier core 413 receives the operands from the operand buffers 411A and 411B and rounding mode information from, for example, a rounding mode store 416. Rounding mode store 416 supplies signals indicating, for example, whether the rounding mode for the current operation is "round toward plus infinity" or "round toward minus infinity." In accordance with the principles of the present invention, one embodiment of the multiplier unit 410 generates results according to the table in FIG. 2, as discussed above. In one embodiment, multiplier core 413 produces results conforming to IEEE Standard 754; certain multiplication operations always produce a NaN result regardless of the rounding mode, thus leading to production of malformed intervals if left unmodified. In the

embodiment shown, however, multiplier decision table logic 415, which controls the result assembler 414, replaces a NaN result with a non-NaN result. In one embodiment, multiplier decision table logic 415 determines whether to substitute a non-NaN result for a NaN result by examining the first operand 411A and the second operand 411B before or during operation of multiplier core 413, knowing that certain multiplication operands produce a NaN under IEEE 754. In such an embodiment, multiplier core 413 could be simplified to not handle cases where a NaN result would be replaced by multiplier decision table logic 415 (making the core IEEE 754 noncompliant), or multiplier core 413 could be bypassed so as not to operate at all in cases where a NaN result would be replaced. In another embodiment that is probably slower because the replacement decision is not made in parallel with the multiplier core's operation, multiplier decision table logic 415 determines whether to substitute a non-NaN result for a NaN result by examining the result produced by multiplier core 413. In yet another embodiment easily within the knowledge of one of ordinary skill in the art, multiplier core 413 produces a non-NaN result for cases where IEEE 754 compliance would otherwise call for a NaN result. In this embodiment, the core in effect self-determines and replaces an IEEE 754 NaN result with a non-NaN result, obviating the need for multiplier decision table logic 415 to make the determination and replacement. In such an embodiment, multiplier core 413 would obviously no longer be IEEE 754 compliant.

[070] Each operand analysis circuit 412A, 412B analyzes the operand in the respective buffer 411A, 411B and generates signals providing information relating thereto, which signals are provided to the multiplier decision table logic circuit 415.

[071] The result assembler 414 receives information from a number of sources, including the operand buffers 411A and 411B, multiplier core 413 and several predetermined

value stores. Under control of signals from the multiplier decision table logic circuit 415, result assembler 414 generates the result, which is provided on a result bus 417. The result bus 417, in turn, may deliver the result to any convenient destination, such as a register in a register set (not shown), for storage or other use.

[072] One of ordinary skill in the art will recognize that an improved multiplier core designed to produce results according to the table of FIG. 2 could be implemented in a floating point multiplier unit that does not use floating point status information encoded within the floating point operands, without departing from the principles of the present invention. One of ordinary skill will realize that the principles of the present invention are also applicable to conventional floating point multiplier architectures that, for example, maintain floating point status information in a global floating point status register, separate from the operands and result. Moreover, it would also be apparent to one of ordinary skill in the art that the NaN result substitutions according to the table in FIG. 2 are not the only possible logical substitutions. Other non-NaN results could be used within the scope of the present invention.

[073] FIG. 5 is a functional block diagram of an exemplary divider unit according to one embodiment of the invention. Generally, the exemplary divider unit 510 receives two floating point operands and generates therefrom a result. In some cases, the exemplary divider unit 510 generates floating point status information encoded in and comprising part of the floating point representation of the result. Since the floating point status information comprises part of the floating point representation of the result, instead of being separate and apart from the result as in conventional divider units, the implicit serialization that is required

by maintaining the floating point status information separate and apart from the result can be obviated.

[074] As shown in FIG. 5, the exemplary divider unit 510 includes two operand buffers 511A and 511B, respective operand analysis circuits 512A and 512B, a divider core 513, a result assembler 514 and a divider decision table logic circuit 515. The operand buffers 511A and 511B receive and store respective operands from, for example, a set of registers (not shown) or memory (not shown) in a conventional manner.

[075] The divider core 513 receives the operands from the operand buffers 511A and 511B, and rounding mode information from, for example, a rounding mode store 516. Rounding mode store 516 supplies signals indicating, for example, whether the rounding mode for the current operation is "round toward plus infinity" or "round toward minus infinity." In accordance with the principles of the present invention, one embodiment of the divider unit 510 generates results according to the table in FIG. 2, as discussed above. In one embodiment, divider core 513 produces results conforming to IEEE Standard 754; certain division operations always produce a NaN result regardless of the rounding mode, thus leading to production of malformed intervals if left unmodified. In the embodiment shown, however, divider decision table logic 515, which controls the result assembler 514, replaces a NaN result with a non-NaN result. In one embodiment, divider decision table logic 515 decides whether to substitute a non-NaN result for a NaN result by examining the first operand 511A and the second operand 511B before or during operation of divider core 513, knowing that certain division operands produce a NaN under IEEE 754. In such an embodiment, divider core 513 could be simplified to not handle cases where a NaN result would be replaced by divider decision table logic 515 (making the core IEEE 754

noncompliant), or divider core 513 could be bypassed so as not to operate at all in cases where a NaN result would be replaced. In another embodiment that is probably slower because the replacement decision is not made in parallel with the divider core's operation, divider decision table logic 515 decides whether to substitute a non-NaN result for a NaN result by examining the result produced by divider core 513. In yet another embodiment easily within the knowledge of one of ordinary skill in the art, divider core 513 produces a non-NaN result for cases where IEEE 754 compliance would otherwise call for a NaN result. In this embodiment, the core in effect self-determines and replaces an IEEE 754 NaN result with a non-NaN result, obviating the need for divider decision table logic 515 to make the determination and replacement. In such an embodiment, divider core 513 would obviously no longer be IEEE 754 compliant.

[076] Each operand analysis circuit 512A, 512B analyzes the operand in the respective buffer 511A, 511B and generates signals providing information relating thereto, which signals are provided to the divider decision table logic circuit 515.

[077] The result assembler 514 receives information from a number of sources, including the operand buffers 511A and 511B, divider core 513 and several predetermined value stores. Under control of signals from the divider decision table logic circuit 515, result assembler 514 generates the result, which is provided on a result bus 517. The result bus 517, in turn, may deliver the result to any convenient destination, such as a register in a register set (not shown), for storage or other use.

[078] One of ordinary skill in the art will recognize that an improved divider core designed to produce results according to the table of FIG. 2 could be implemented in a floating point divider unit that does not use floating point status information encoded within

the floating point operands, without departing from the principles of the present invention. One of ordinary skill will realize that the principles of the present invention are also applicable to conventional floating point multiplier architectures that, for example, maintain floating point status information in a global floating point status register, separate from the operands and result. Moreover, it would also be apparent to one of ordinary skill in the art that the NaN result substitutions according to the table in FIG. 2 are not the only possible logical substitutions. Other non-NaN results could be used within the scope of the present invention.

[079] One of ordinary skill in the art will recognize that the results chosen to substitute for the IEEE Std. 754 NaN results in the illustrated embodiments, as well as the conditions and circuits chosen to produce the improved results, could easily be altered without departing from the principles of the present invention. For example, one of ordinary skill could easily design circuits and methods that produce a negative zero result if an infinity operand is multiplied with a zero operand having the opposite sign, with a round toward plus infinity rounding mode. Such alterations improve over the IEEE 754 result when the operation is used in interval arithmetic calculations and fall within the scope of the present invention.

[080] It will be appreciated that a system in accordance with the invention can be constructed in whole or in part from special purpose hardware or a general purpose computer system, or any combination thereof, any portion of which may be controlled by a suitable program. Any program may in whole or in part comprise part of or be stored on the system in a conventional manner, or it may in whole or in part be provided to the system over a network or other mechanism for transferring information in a conventional manner. In

addition, it will be appreciated that the system may be operated and/or otherwise controlled by means of information provided by an operator using operator input elements (not shown) which may be connected directly to the system or which may transfer the information to the system over a network or other mechanism for transferring information in a conventional manner.

[081] Those skilled in the art will appreciate that the invention may be practiced in an electrical circuit comprising discrete electronic elements, packaged or integrated electronic chips containing logic gates, a circuit utilizing a microprocessor, or on a single chip containing electronic elements or microprocessors. It may also be provided using other technologies capable of performing logical operations such as, for example, AND, OR, and NOT, including but not limited to mechanical, optical, fluidic, and quantum technologies. In addition, the invention may be practiced within a general purpose computer or in any other circuits or systems as are known by those skilled in the art.

[082] The foregoing description has been limited to specific embodiments of this invention. It will be apparent, however, that various variations and modifications may be made to the invention, with the attainment of some or all of the advantages of the invention. It is the object of the appended claims to cover these and such other variations and modifications as come within the true spirit and scope of the invention.